Worksheet 2: Recursion, Data Structures, and Search

Due: Before midnight, 17 October 2005.

Total marks: 50

Question One (10 marks)

Implement Merge sort. Assume that the input is a list, and the output is a list. Merge sort splits the input list roughly in half, sorts each half recursively, and merges the resulting sorted sublists. Note that a version of `merge` appears on the class Web page, but it may have bugs in it.

Hand in well documented source code, and a file in which you show the testing you did.

Question Two (20 marks)

We saw a simple implementation of binary search trees in lecture. We will represent a binary search tree using `tree(Key,Left,Right)` for non-empty trees (where `Key` is the search key, and `Left` and `Right` are binary search trees (subtrees)). An empty binary tree will be represented by the atom `empty`. [1]

We can write a program to check if a key is in a BST as follows:

```
% memberBST(+Key,+Tree)
% true if the Key is in the Tree

memberBST(Key,tree(Key,_,_)).
memberBST(Key,tree(K2,L,_)) :-
  Key < K2,
  memberBST(Key,L).
memberBST(Key,tree(K2,_,R)) :-
  Key > K2,
  memberBST(Key,R).
```

In this implementation, we are assuming that all keys will be unique, so we don't have to worry about equality.

1. Draw a search tree for `memberBST(3,tree(2,tree(1,empty,empty),tree(3,empty,emtpy)))`.
2. Draw a proof tree for `memberBST(3,tree(2,tree(1,empty,empty),tree(3,empty,emtpy)))`.
3. Implement `insertBST(Tree1, Key, Tree2)` which relates a given `Tree1` (the original tree) and a given `Key` to the tree `Tree2` which is the result of inserting `Key` into `Tree1`. Note that `Tree1` does not change, but rather, we are building a slightly modified tree `Tree2`. This approach is very common in Prolog.

   As an example:

[1] There are lots of other ways to represent BSTs. For an interesting alternative, look at PiP, Chapter 7. The code there will not be useful here, though.

```
?- insertBST(empty,2,T1).
T1 = tree(2,empty,empty)
yes
?- insertBST(empty,2,T1), insertBST(T1,1,T2).
T1 = tree(2,empty,empty)
T1 = tree(2,tree(1,empty,empty),empty)
yes
```

4. Write a program `removeBST(Tree1, Key, Tree2)` that "removes" a given element from a given binary search tree. Note that the original tree is not changed; rather, a new tree is constructed with the given key gone. We will do this in steps.

   As an example:

```
?- removeBST(tree(2,tree(1,empty,empty),empty), 2, T).
T = tree(1,empty,empty)
yes
```

   (a) Write a program that, given a BST, removes the biggest key, and returns the BST containing all the remaining keys. In other words, `deleteMax(+Tree1,?Key,?Tree2)` is true when `Key` is the biggest key in `Tree1`, and `Tree2` contains the same keys as `Tree1` except that `Key` is missing. Procedurally, given `Tree1`, you can use this to return two things: a `Key` and the resulting `Tree2`.

   Hint: the biggest key in a BST is found by stepping recursively into the right subtree. (Draw a picture of the situation! It's easier drawn than described.)

   (b) Write `removeBST(Tree1, Key, Tree2)` using `deleteMax(+Tree1,?Key,?Tree2)`.

   Once the key you are looking for (k) is located, you can replace k with the largest key k' in the tree that is smaller than k; this maintains the binary search tree property.

Keep your relationships well-defined, and your implementations short. Rules that are more than handful of lines long are more difficult to write; feel free to write auxiliary rules.

Also, my descriptions above are sometimes procedural. Try to think in terms of relationships as well as in terms of procedure. Both aspects are helpful for different purposes. Here, we want to keep in mind the property of BSTs, namely that the keys are organized in a specific way. It will help!

Question Three (20 marks)

Back in the olden days, people used cash money to trade for goods and services. This was before the age of debit cards. When you wanted to purchase something, you handed to cashier some cash in exchange for the goods. In the case that you did not have exact change, you were allowed to give more money to the cashier, who was expected to return to you the difference between the cash you gave, and the value of the goods. This difference is often called "change." (By the way, you were not generally allowed to hand over less cash than the goods were worth.)

We will work with Canadian currency. We will limit our application to the use of pennies, nickels, dimes, quarters, loonies, toonies, fives, tens, twenties, fifties, and one hundred dollar bills.

We can determine by subtraction the amount of cash to be returned to the buyer. For example, if a buyer hands over $10.00 for goods worth $8.95, the change is $1.05. However, the change must be made up of the currency elements. The change can be composed of a loonie and a nickel, or 4 quarters and a nickel, or three quarters and three dimes. You may find it simpler to deal with integers, count the number of "cents" rather than floating point number and fractions of "dollars." See my example below.

Write a Prolog program to compose change. It should return a sensible (and correct, of course) description of the change in terms of a list. But it should also backtrack to allow alternate possibilities. Maybe the cashier has run out of loonies, and has to substitute four quarters. For example:

```
?- change(185,L).

L = [number(loonie, 1), number(quarter, 3), number(dime, 1)] ;

L = [number(loonie, 1), number(quarter, 3), number(nickel, 2)]

yes
```

In the example, I have made use of a list and a simple compound term to represent the change. You may use this example, or some device of your own choosing.

Don't worry about sorting the elements, or returning only unique answers. It's enough, for now, to find a variety of ways of making change. The issue here is to learn to use Prolog's non-determinism to solve problems.

I haven't broken down the problem as I did with Question One. Break the problem in relationships that can be written as simple rules. As a guide, my solution used 3 sets of rules, and is about 20 lines long (not including comments, or the data for the currency).

Hand in a well documented source code file, and a file demonstrating the testing you did.

Submission

Submit your work using the EHandin facility. Please take care to submit code as text files (not Word docs, or rtf or ...). Your code should be well-documented, including instructions on how to run the code, and you should show the results of running your program on some test data. Discussion questions may be answered using other common file formats (MS Word, PostScript, PDF, etc). If the marker cannot open your file, or cannot simply test your program, you will receive no grade, and will be asked to resubmit.

Late worksheets will be not accepted.